Making Android Apps Safer: Improving the Portability of Application Behaviour Monitoring Tools

Author: Karyotakis Ioannis

Student ID: 8210055

Degree Program: Department of Management Science and Technology

Supervisors: Diomidis Spinellis & Nikolaos Alexopoulos

Athens University of Economics and Business

July 5, 2025

Abstract

This thesis addresses the portability challenges of kernel-level tracing in diverse Android devices, emphasising behavioural reconstruction via the *SliceDroid* approach. Kernel-level tracing is crucial for security, performance, and application analysis but is impeded by Android fragmentation, vendor customisations, and kernel restrictions. The research evaluates tracing tools like *ftrace* and *kprobes*, identifies device-specific limitations, and proposes solutions to improve *SliceDroid*'s portability. Additionally, the study expands the tracing mechanism to cover regular files and databases, enhancing behaviour monitoring and security analysis. Practical contributions include an improved tracing setup, streamlined device-specific configurations, and a robust, plug-and-play tool for comprehensive behavioural profiling in Android systems.

Η παρούσα πτυχιακή εργασία εξετάζει τις προκλήσεις στη φορητότητα του *tracing* σε επίπεδο πυρήνα (*kernel-level tracing*) σε διάφορες συσκευές Android,με έμφαση στην ανακατασκευή συμπεριφορών μέσω της προσέγγισης *SliceDroid*. Το *tracing* σε επίπεδο πυρήνα είναι απαραίτητο για την ανάλυση ασφάλειας, απόδοσης και εφαρμογών, ωστόσο η υλοποίησή του παρεμποδίζεται λόγω του κατακερματισμού του Android, των τροποποιήσεων των κατασκευαστών και των περιορισμών στον πυρήνα. Η έρευνα αξιολογεί εργαλεία *tracing* όπως τα *ftrace* και *kprobes*, εντοπίζει περιορισμούς συγκεκριμένων συσκευών και προτείνει λύσεις για τη βελτίωση της φορητότητας του *SliceDroid*. Επιπλέον, η μελέτη επεκτείνει τον μηχανισμό *tracing* ώστε να καλύπτει και κανονικά αρχεία καθώς και βάσεις δεδομένων, ενισχύοντας την παρακολούθηση συμπεριφοράς και την ανάλυση ασφάλειας. Οι πρακτικές συνεισφορές περιλαμβάνουν βελτιωμένη διαδικασία εγκατάστασης *tracing*, απλοποιημένη ρύθμιση ανά συσκευή, καθώς και ένα εύχρηστο και αξιόπιστο εργαλείο για ολοκληρωμένη ανάλυση συμπεριφοράς σε συσκευές Android.

Contents

1 Introduction **4**

2 Related Work **7**
2.1 Impact of Android Fragmentation 7
2.2 Behavioural Reconstruction in Android 8

3 Background **10**
3.1 Android Architecture and Kernel Fragmentation 10
3.2 Kernel-Tracing Mechanisms 10
3.3 Portability Obstacles 13
3.4 SliceDroid as a Baseline for Portable Behavioural Reconstruction 16

4 Methodology **19**
4.1 Limitations of SliceDroid and Experimental Setup 19
4.2 Behavioral Reconstruction Portability Enhancement 20
4.3 Databases and Network Monitoring 22
4.3.1 Locating common SQLite databases 22
4.3.2 Network-level instrumentation 23
4.3.3 Demonstration APK and preliminary findings 23

5 Results **24**
5.1 Device Nodes Mapping Results 24
5.2 Regular file access and network traffic integration Results 26
5.3 Overall contribution 28

6 Threats to Validity **31**

7 Conclusion **32**

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors, Nikolaos Alexopoulos and Diomidis Spinellis, for their continuous support, valuable guidance, and constructive feedback throughout this thesis.

I also wish to thank my friends and colleagues, Foivos and Vaggelis, for their encouragement and useful discussions.

I am deeply thankful to my family for their unwavering support, patience, and encouragement throughout my studies and the writing of this thesis.

Last but not least, I would like to acknowledge Ioannis & Aikaterini Samaras Non Profit Partnership for their continuous support during my undergraduate studies.

[Ioannis Karyotakis]

[July 5, 2025]

Chapter 1

Introduction

Android is the most widely adopted operating system globally for end-user devices, underpinning billions of smartphones and tablets [1]. The pervasive nature of Android makes the enhancement of system observability increasingly critical, especially concerning security, performance optimisation, and application analysis.

At its core, the Android kernel is derived from the upstream Linux Long Term Supported (LTS) kernel [2]. This heritage enables Android to leverage mature and powerful kernel tracing mechanisms from Linux, such as *ftrace* [3], *kprobes* [4], and the Linux Trace Toolkit Next Generation (LTTng) [5]. Kernel-level tracing is particularly valuable as it provides essential capabilities for detecting malware [6], analysing over privileged applications [7], optimising system performance [8, 9], and facilitating reverse engineering [10]. However, deploying these tracing techniques faces significant challenges due to dependencies on device-specific kernel characteristics.

In parallel, Android's widespread adoption has made the platform a target for increasingly sophisticated attacks. Prior studies have shown the relevance of supply chain attacks [11, 12], privacy breaches [13] or malicious repackaged apps in third-party app marketplaces [14]. To address these threats in the wild, dynamic analysis can be instrumented.

Given Android's ubiquity and diverse threat surface, leveraging kernel-level observability has become essential. Dynamic analysis frameworks increasingly harvest kernel traces, for example, syscall logs or eBPF events to capture the raw evidence of an app's runtime behaviour [15, 16, 17]. Lifting these low-level events into concise, semantically rich actions enables security monitors to surface malicious logic even when traditional static inspection fails, providing an out-of-the-box defences.

One of the analysis techniques that use kernel level tracing is behavioural reconstruction. Android behavioural reconstruction refers to the process of analysing and recreating the actions and interactions of an Android application or system. This is not by any means trivial, as there is a semantic gap between low-level system-call invocations and high-level Android-specific behaviour [18].

We focus on the SliceDroid's reconstruction approach because of its simplicity, low overhead, and extensibility make it a strong candidate for cross-device portability [19]. SliceDroid reconstructs process behaviours from Linux kernel traces by capturing I/O events and mapping them to high-level behaviours using device node mappings and kernel instrumentation technologies like *kprobes*. However, the practical deployment of SliceDroid across different

Android devices is hindered by device-specific variability, such as restricted access to the kernel symbol table (/proc/kallsyms).

To mitigate these issues, this research empirically examines tracing behaviours across multiple Android devices, identifies key limitations affecting portability, and describes the applied solutions to address these challenges. Major contributions include empirical insights into kernel-level tracing portability and practical solutions that enhance SliceDroid's applicability and resilience.

Historically, Android provided a standardised tracing approach through the *systrace* tool. However, recent Android versions have transitioned to *Perfetto*, Google's official tracing framework tailored primarily for performance diagnostics. Perfetto integrates tightly with Android's platform architecture and supports devices adhering to the Generic Kernel Image (GKI) standard, enhancing portability. Despite these improvements, Perfetto remains primarily focused on system and application performance monitoring rather than security-focused or arbitrary kernel-level tracing. Both tools use ftrace under the hood, as systrace is built on top of ftrace [20] and perfetto integrates closely with it too [21].

In contrast, tracing mechanisms such as *kprobes* offer the ability to monitor almost any kernel function dynamically, making them particularly suitable for fine-grained behavioural profiling and security analysis. However, their portability across Android devices is limited by extensive kernel-level variability, commonly known as *Android fragmentation*. Prior to GKI, vendor-specific customisation could constitute up to 50% of the Android kernel codebase [22], leading to significant disparities in kernel configuration, symbol availability, and overall behaviour.

Even after the adoption of the GKI standard, significant inconsistencies persist. Variations in SELinux policies, symbol table visibility, kernel configuration options, and tracepoint availability across manufacturers continue to challenge the creation of universal kernel-level tracing solutions.

Addressing these will facilitate streamlined malware detection, accelerate robust monitoring deployment, and enhance security tooling throughout the Android ecosystem. The preceding discussion highlights unresolved gaps in *portability* and *observability*. These gaps motivate two research questions.

Research Question (RQ-1)

How can we design a device-agnostic kernel-tracing and behaviour-reconstruction pipeline that works across diverse Android devices, enabling consistent profiling of high-level behaviours?

Research Question (RQ-2)

How can we trace kernel-visible behaviours together with regular file accesses and network traffic to support comparative security and privacy analysis of Android apps?

Together, **RQ-1** tackles the challenge of *portable observation*, while **RQ-2** focuses on converting that observation into actionable traces for comparative security and privacy analysis.

This thesis is organised as follows. Chapter 2 surveys related work; Chapter 3 provides the necessary background; Chapter 4 details our methodology; Chapter 5 presents the results of our cross-device evaluation; Chapter 6 notes some threats to validity and Chapter 7 concludes the thesis.

The next chapter reviews previous efforts to curb Android fragmentation and to reconstruct application behaviour, while Chapter 3 supplies the technical foundations on which those efforts rest—letting readers explore the theory after first seeing the problem and the motivation for this thesis.

Chapter 2

Related Work

Related work falls into two axes: (i) how fragmentation manifests and (ii) how researchers rebuild high-level behaviour from low-level traces.

2.1 Impact of Android Fragmentation

Prior studies quantify how kernel-level fragmentation hampers tracing and malicious behaviour analysis. From early on the life of android project, the differences among devices was an important topic. Notably, software engineers face challenges during development [23] and during testing [24], to develop cross-device functionality. Moreover, Han et al. [16] used Latent Dirichlet Allocation (LDA) to compare bug topics extracted from Android bug reports, focusing on devices from HTC and Motorola. Distinguishing between common and vendor-specific topics highlighted the lack of portability in the Android ecosystem. They found that even topics shared across vendors differed in their keywords, indicating that devices from different manufacturers encountered different issues. For example, the HTC calendar topic includes the terms "2.2," "running," and "reminder" (bugs affecting the Desire running Android 2.2), whereas Motorola's calendar topic features "droid," "milestone," and "outlook" (issues on the Droid and Milestone lines). Wei et al. [25] distinguished 5 major reasons for the fragmentation, with the platform API evolution and problematic hardware driver implementation being the most significant.

Numerous studies have highlighted the impact of such fragmentation. For instance, Liu et al. [7] analysed privacy leaks across different Android firmware variants, revealing substantial behavioural inconsistencies. As expected, that situation creates security problems. Zhou et al. [26] reported that the openness and fragmentation of android, has resulted in significant security implications due to the customisation of devices. Nguyen-Vu et al. [27] highlights that machine-learning approaches that build vectors by decompiling apps and extracting components across different OS versions can inadvertently omit critical features needed for accurate detection.

2.2 Behavioural Reconstruction in Android

Numerous studies have aimed to reconstruct high-level Android behaviours from low-level system information, employing various methodologies:

Dynamic Taint Tracking. Approaches such as DroidScope utilise taint analysis to extract detailed execution traces from both native and Dalvik instructions. It profiles activities at the API level and identifies information leakage across Java and native layers, using a QEMU-based CPU emulator [16]. Another prominent system, TaintDroid, implements granular variable tracking within the Dalvik VM interpreter, enabling real-time monitoring of how third-party apps handle sensitive user data. TaintDroid tags data flows within the VM and performs analysis to identify malicious or unintended data handling [28].

Syscalls and IPC-based Solutions. Dagger reconstructs Android application behaviours by generating provenance graphs from system calls, Binder transactions, and process-level details. It identifies behaviours by comparing these provenance graphs against a repository of sensitive behaviour patterns, derived from comprehensive analysis of the Android framework internals. Although Dagger avoids extensive VM instrumentation or kernel modifications, it incurs a runtime overhead of approximately 63% [18]. In contrast, CopperDroid captures system call events by intercepting CPU privilege-level transitions in an otherwise unmodified Android environment running on a customised emulator. It translates Binder communications into human-readable representations of inter-process communication (IPC) [17].

Mapping Android APIs to Syscall Sequences. The approach by Nisi et al., which inspired SliceDroid, initially involves generating a dataset correlating Android API calls with their associated syscalls. Subsequently, it maps generic syscall sequences back to specific API calls. To facilitate this, all public methods within the Android Open Source Project (AOSP) framework are automatically instrumented at their entry and exit points [29].

The behavioural reconstruction workflows described above suffer from one or more of the following limitations:

- They depend on the intermediate layers of the OS and their implementation details
- They have impact on the system, in terms of overhead
- They can't be deployed in a real-world scenario easily, as they demand VM instrumentation or kernel modifications

SliceDroid achieves to mitigate these obstacles but ,also, because it traces device accesses directly in the kernel layer, overcomes existing shortcomings of Android 12's visual privacy indicators for camera and microphone used by non-system apps. Android 12 introduced those indicators by inspecting relevant API calls and showing a status-bar icon for at least five seconds [30], but they don't cover system apps and can be bypassed by other app by e.g. framework bugs [31]. By instrumenting at the kernel level, SliceDroid ensures comprehensive, tamper-resistant monitoring of all device accesses.

In short, while reconstruction techniques are increasingly sophisticated, they inherit the portability (i.e. kernel modifications) and complexity headaches(i.e vm instrumentation) just described. Chapter 3 therefore steps back to the architectural ground truth and the tracing machinery on which most solutions (including SliceDroid) build.

Chapter 3

Background

Chapter 3 provides the technical backdrop that underpins every experiment reported later: first the Android kernel landscape, then the tracing primitives available, the obstacles they face in the wild, and finally the SliceDroid baseline.

3.1 Android Architecture and Kernel Fragmentation

Android is built on top of the Linux kernel, with its system architecture comprising layers that extend from the kernel space up to application-level components [2]. The platform consists of the Linux kernel, hardware abstraction layer (HAL), native libraries, Android Runtime (ART), and framework services [32]. While the core components are open-source and standardised through the Android Open Source Project (AOSP), the platform has historically suffered from significant fragmentation. Kamran and Nisar [33] state that android fragmentation can be classified at kernel level, api level and hardware level. Kernel level fragmentation is the main focus in the thesis.

Before the introduction of the Generic Kernel Image (GKI) initiative, Android device manufacturers maintained heavily customised kernel versions with out-of-tree drivers and proprietary modifications. Vendors could take the Android Common Kernel (ACK) and implement extensive changes to suit their hardware needs, leading to significant fragmentation across devices [22].

With the sources of fragmentation now mapped out, we turn to the tracing tools that must operate across this diverse landscape.

3.2 Kernel-Tracing Mechanisms

Tracing in the linux kernel has more than 25 years of history. Many solutions have been created either for desktop or android systems, offering distinct benefits. Thus, it makes sense to overview the most well-known tools.

Firstly, in 2000 Yoghmour and Michel Dagenais developed the Linux Tracing Toolkit. The toolkit consisted of a kernel patch to log the events, a kernel module to store them in a buffer, a trace deamon and , lastly, a data decoder to present a human-readable format [34]. Other than logging and presenting the trace, it even offered plotting of the tracing

results [35]. Its successor, Linux Tracing Toolkit Next Generation (LTTng), had the same goals of minimal performance overhead and architecture independence [36]. Moreover, the capability of tracing user space was offered, however, as in LTT, only privileged users can use it and a single tracing source is available [37].

Another widely used tracing mechanism for kernel is ftrace. It was released in 9 October 2008 with kernel version 2.6.27, as a very simple function tracer and was added to the /debugfs/tracing [38]. Later in the same year, on 25 December 2008, with kernel version 2.6.28 tracepoints were introduced [39], as an improvement over the existing static tracepoints i.e. kernel markers [40]. Before 4.1, all ftrace tracing control files were within the debugfs file system. In more recent versions, Ftrace uses the tracefs file system to hold the control files [41]. The reason is that there were compaints about tracing being too dependent on the debugfs, while many systems could not mount it due to security reasons [42]. Although ftrace is commonly referred to as a function tracer, it is more accurately a framework encompassing a variety of tracing utilities. It can use `TRACE_EVENT` for static instrumentation or Kprobes to dynamically hook into various parts of the kernel. From early on it was used for low-level tasks like measuring the duration of functions' execution [43].

A tool with different uses cases than the previous two is Perf. Perf was initially used an an interface for the performance counters subsystem in Linux and does sampling and profiling [44]. It also supports tracepoints, kprobes, and uprobes [45]. It is suitable answer practical questions such as cache-friendliness of the code [46].

SystemTap is one of the most well-known tracing tools that is not part of the Linux mainline kernel. One of the main reasons is that, for kernel instrumentation, it needs installation of the debugging symbols for the kernel currently running [47]. SystemTap scripts are the foundation of each session and follow the same syntax and semantics as the C programming language [48]. SystemTap then translates the script to C, loads the module, enables probes and executes handles when needed [49]. In that sense, SystemTap is a system that facilitates the creation of monitoring tools [50].

eBPF (extended Berkeley Packet Filter) started evolving significantly around 2014, when it was integrated into the Linux kernel. It is widely used for observability, network security, behavioral security etc. [51]. EBPF is currently the most trending and rapidly evolving tracing technology in the Linux ecosystem, and for good reason. EBPF drastically improves processing by being JIT compiled, making it run as efficiently as natively compiled kernel code or as code loaded as a kernel module [52]. Furthermore, eBPF programs are verified to not crash the kernel and can be used to modify or add functionality and use cases to the kernel without having to restart or patch it [53]. Unless unprivileged eBPF is enabled, any process that wants to load eBPF programs into the Linux kernel must either have root privileges or possess the `CAP_BPF` capability. A more common approach to develop eBPF scripts is to use a compiler suite like LLVM to compile pseudo-C code into eBPF bytecode. Ebpf allows to change the behavior of the kernel without creating kernel modules. eBPF programs are event-driven and execute when triggered by specific hook points in the kernel or an application. These predefined hooks include function entry and exit, kernel tracepoints, kprobes and more.

Many of the standard Linux tracing tools (like perf, eBPF, ftrace) can be used in Android — especially on rooted or engineering/debug builds — but Android also includes its own tracing tools and APIs tailored to mobile environments and app developers. Basically, they

function as a wrapper over existing data sources and tracers. Perfetto has emerged as the de facto framework for low-level tracing on Android devices since Android 10, replacing legacy tools like systrace while offering enhanced cross-layer observability [54]. As a unified tracing stack, it enables simultaneous analysis of both userspace processes and kernel-space events. Userspace monitoring happens through Atrace hooks. Atrace injects trace points to native and managed apps (userspace) and writes the output to the ftrace buffer (/sys/kernel/debug/tracing/trace_marker) [55]. On the other side, perfetto's close integration with ftrace, allows it to offer kernel tracing capabilities. Moreover, it captures data from the filesystem data sources i.e./proc/<pid>/status and tools like Heapprofd that tracks heap allocations and deallocations of an Android process [56]. With v49.0 from January 7, 2025 Perfetto supports basic kprobes functionality with ftrace. This enhancement allows Perfetto to utilise kprobes for kernel tracing, providing more granular insights into kernel function calls and performance metrics [57]. Despite the recent addition, tracing frameworks for android come with specific limitations. Specifically, according to the android documentation, ftrace offers more functionality than can be directly enabled by systrace or atrace. Thus it is can offer extra capabilities e.g performance debugging [20]. Moreover, in the perfetto framework, ftrace is used mainly for scheduling events, system calls and frequency scaling. However, only the syscall number is recorded and the arguments are not stored to limit the trace size [58]. Ftrace can offer extensive capabilities for dynamic instrumentation and fine-grained kernel analysis. As a consequence, low-level tools are more appropriate for behavioral reconstruction, security-focused research (e.g., permission misuse, system call anomalies), dynamic kernel instrumentation, custom tracepoints, runtime inspection and syscall hooking.

All of the tracing systems described are based on some of the above tracing mechanisms:

Static kernel Tracepoints. Static instrumentation using predefined kernel hooks. Efficient, stable among kernel versions and used from many tools.

Kprobes. Dynamic instrumentation of kernel functions at runtime. Used for tracing deep internals. Kprobes are indeed powerful but have minimum safety checking and have almost non-existent portability because they need kernel addresses and specific kernel symbols to implement instrumentation [59]. Additionally, similarly to Kprobes, Kretprobes can be used to access the return value of a function.

Fentry/Fexit. Fentry was introduced alongside BPF trampolines in Linux kernel version 5.5. In modern kernels, fentry and fexit are the preferred mechanisms for tracing function entry and exit. These probes can be attached to any kernel function that contains a sufficient number of NOP (no-operation) instructions. When attached, a BPF trampoline—architecture-specific, dynamically generated machine code—is created to invoke the corresponding eBPF program [60]. Equivalent code can be written inside a kprobe or fentry type program [61]. These eBPF program types are more efficient than kprobes and give access to the input parameters to the function, which kretprobe does not.

Perf events. Instruments "events", which serve as a unified interface to various kernel instrumentation sources—such as hardware performance counters, software-generated events

and more[62]. There are also tracepoint events, implemented by the kernel ftrace infrastructure [63].

BPF Hooks. BPF hooks combine existing mechanisms like tracepoints with bpf-specific attach points i.e. for network but with difference in speed and safety guarantees. BPF hooks represent predefined attachment points within the Linux kernel where eBPF programs can be safely executed. These hooks span a variety of subsystems, including network-layer hooks such as XDP, device-level interfaces like HID-BPF, and mechanisms previously discussed, including kprobes and perf events [64]. Central to the secure execution of eBPF programs is the eBPF verifier, a core component of the BPF subsystem. The verifier statically analyzes eBPF bytecode to ensure compliance with a strict set of safety constraints, such as prohibiting arbitrary memory access and enforcing bounds checking [65]. While eBPF-based kprobes and other hook types offer a safer and more portable alternative to traditional dynamic instrumentation techniques, their functionality is inherently constrained by the verifier's safety model. Consequently, they may not fully replicate the flexibility offered by lower-level mechanisms, but they provide a reliable and robust framework for dynamic instrumentation in most practical scenarios.

Table 3.1: Portability of Linux and Android Tracing Tools

Tool	**Portability**	**Justification**
ftrace	★★★★★	Built into the kernel; no dependencies; widely supported
perf	★★★★	Widely available; depends on perf event support and hardware counters
eBPF	★★★★	Requires kernel $\geq$ 4.9, BTF for some features
SystemTap	★★	Needs debug symbols; less portable across systems
LTTng	★★★	Requires kernel modules; works best on debug builds
Perfetto	★★★★	Native to Android 10+; user/debug build access varies
Systrace/atrace	★★★★	Default on Android; limited kernel access on user builds

Even the richest tracing stacks stumble when faced with real-world device diversity; the next section catalogues those stumbling blocks.

3.3 Portability Obstacles

Implementing kernel tracing tools across the Android ecosystem comes with portability challenges, due to device and kernel variations:

Kernel Version Differences. Android devices run various Linux kernel versions, so the availability and signatures of kernel functions differ. A kprobe-based tracer that works on one version may break on another if the target function's name or arguments change, or if the function is removed. In practice, these internal changes require tracing tools to be updated or adapted for each kernel version. A more stable alternative is the tracepoints of the kernel, as they generally remain stable across kernel versions

OEM Customizations. Manufacturers frequently apply substantial customizations to their device kernels, altering both their behavior and structure. Prior to the introduction of Google's Generic Kernel Image (GKI), as much as 50% of an Android kernel's code could consist of out-of-tree vendor-specific modifications. Despite GKI's goal of unifying core kernel implementations, significant vendor-specific kernel alterations persist, meaning that tracing interfaces and function behaviors can vary substantially between devices. A tracing methodology effective on a Pixel device may not be directly applicable to a Samsung or OnePlus kernel due to differences like variations in the kernel's symbols table.

For example, during instrumentation testing for the results presented in this thesis, a particular complication arose on the OnePlus Nord CE4 device, involving kernel symbol ambiguities. Specifically, the kernel symbol table (`/proc/kallsyms`) listed both strong (`T`) and weak (`w`) symbols for critical kernel functions like `vfs_write` and `vfs_read`, the weak symbols distinctly annotated with an ELF symbol-version namespace `[oplus_chg]`, corresponding to the device-specific kernel module `oplus_chg`. This dual entry resulted in the kernel tracing facility (ftrace) kprobe resolver rejecting symbolic probes due to ambiguity, indicating that the symbols were not unique. To resolve this, the exact memory addresses of the strong symbol, explicitly obtained from `/proc/kallsyms`, had to be directly specified in the kprobe definitions. Moreover, the `$argN` shorthand commonly utilised by higher-level tracing tools such as `perf` or `bpftrace` was unsupported by the native ftrace kprobe parser, necessitating direct specification of ARM64 argument registers (`%x0`, `%x1`, `%x2`, etc.). Through this explicit disambiguation, accurate and reliable kernel function tracing was achieved on the OnePlus hardware. Conversely, tracing the same VFS functions using symbolic references and the `$arg` shorthand posed no difficulties on devices such as the Nothing 2a, Pixel 9, and Samsung A15.

SELinux Restrictions. Android's security policies (SELinux) often restrict access to kernel tracing facilities like ftrace or kprobes. Many production devices do not mount or expose the debugfs tracing directory at all in normal (user) mode. In fact, devices launching with Android 12+ must never mount debugfs on user builds [66]. Only in development (userdebug) builds or through privileged system components (e.g. the dumpstate tool) can debugfs be accessed. Moving tracefs outside of debugfs allowed android devices to use it excessively, as it-is or as a foundation for systrace and perfetto. Of course, in production builds still some features are restricted but offer valuable insights. These SELinux-enforced restrictions mean that even if the kernel supports tracing there are several limitation that should be taken into consideration, For example, when running Perfetto, If the output file is not under /data/misc/perfetto-traces, tracing will fail due to SELinux [67]. Build-time kernel config options for tracing vary across Android kernels. Some devices ship with certain tracing features

disabled. Setenforce utility can be used to switch between enforcing and permissive mode. To change to permissive mode, enter the `setenforce 0` command [68]. For example, an AOSP kernel for Pixel phones initially had kprobes and related ftrace events turned off in its default config (missing CONFIG_KPROBE_EVENT, CONFIG_FTRACE_SYSCALLS, etc.). If a kernel is built without a given tracer (or with it as a module that isn't loaded), tools depending on that feature simply won't work on that device. Thus, one must account for config flags like CONFIG_FTRACE or CONFIG_KPROBES potentially being unset on certain Android kernels.

Stripped or Missing Kernel Symbols. Many Android kernels hide or omit kernel symbol information, hindering tools that rely on those symbols. In particular, the kernel symbol table (/proc/kallsyms) is often restricted: by default Android does not expose kernel symbol addresses to userspace. For example, kptr_restrict is used for hiding kernel pointers from unprivileged user [69]. (Instead, addresses may show as 0x00000000, etc., unless you have root and disable the restriction.). To solve that, `echo 0 > /proc/sys/kernel/kptr_restrict` can be used to disable the configuration. This is a security measure to make kernel exploits harder, but it also means a tracer can't easily lookup function addresses by name on a stock device. Additionally, some kernels may be built without certain symbols at all (for example, not including un-exported symbols) to further reduce what information is available. All of this makes porting tracing tools tricky, as symbol resolution can fail on some devices.

Tracepoint Variability. The set of static tracepoints available in the kernel is not uniform across Android kernels. Tracepoints are added and changed over time: a given event present in a newer mainline kernel may not exist in older kernel versions used by some devices. For instance, before 4.16, the tool tcplife used the TCP state tracepoint `tcp:tcp_set_state` [70]. In Linux 4.16+, that tracepoint was removed and replaced with `sock:inet_sock_set_state`, breaking any tool that expected the old event[71]. Moreover, Android vendors can introduce custom tracepoints via the "vendor hook" mechanism in the Android Common Kernel – essentially adding their own tracing hooks in places like process exit or scheduler code [72]. As a result, one device's kernel might lack trace events that another device has and vice versa. Tracing tools must be aware of these differences in available events and gracefully handle cases where an expected tracepoint is absent or differently named.

Kernel Structure Layout Variability. Even when kernel functions or tracepoints remain available, the internal layout of kernel data structures (e.g., `task_struct`, `sock`, `sk_buff`) can vary significantly between kernel versions and vendor builds. These layout differences include field reordering, size changes, or even field removal. Tracing tools that read structure fields using hardcoded offsets or rely on outdated header definitions (e.g. BCC need kernel headers of host machine) may misinterpret memory or crash. This issue is especially problematic when using low-level techniques like `bpf_probe_read()`. For example, in commit `2f064a5` of the v6.15-rc1 linux kernel, `task_struct::state` was renamed to `task_struct::__state` [73]. As a result, libbpf-tools and bcc-tools broke [74].

Modern tools based on BPF CO-RE (Compile Once, Run Everywhere) solve this by building in a single binary, bringing together the BTF type information, libbpf and the compiler.

BTF, BPF Type Format, is a metadata format designed for debug information related to BPF programs [75]. Libbpf relies on BTF information provided by the running kernel, which exposes this data through /sys/kernel/btf/vmlinux. This file contains metadata describing all kernel types and structure layouts[76].

Understanding these obstacles clarifies why SliceDroid makes the design choices it does, using just a handful of kernel hooks. § 3.4 summarises that design as the springboard for our own extensions.

The mechanism that we will cover in the rest of the thesis, SliceDroid, is a methodology that maps low-level kernel behavior to a higher-level (userspace) activity, e.g. access to audio or camera. To achieve that, it uses the ftrace virtual filesystem and specifically kprobes and tracepoints to monitor kernel activity. Then, it matches the file identifier (r_dev) of devices node to the r_devs returned from the trace file, to identify the usage of a functionality like nfc.

3.4 SliceDroid as a Baseline for Portable Behavioural Reconstruction

This chapter **summarises the original SliceDroid design** highlighting the assumptions, tracing stack and IPC-slicing algorithms on which my own work builds. No new contribution is introduced here. First, we need to define how SliceDroid works. Is is based on two assumptions, as:

- Interactions with hardware drivers happen through the linux virtual filesystem (vfs), with vfs kernel functions like vfs_write, vfs_read and vfs_ioctl.
- The information among Android Layers flows by Inter-Process communication, e.g. binder.

SliceDroid reconstructs inter-process communication using two PID-based slicing algorithms: a forward IPC slice and a backward IPC slice. The forward slice starts with a known source process and scans the trace sequentially. When it encounters an IPC event from a tracked PID, it adds the destination PID to a set and includes subsequent `write events` from those PIDs in the output. The backward slice processes the reversed trace, starting from a target process. It tracks `read events` and IPC senders that interact with already-known PIDs, gradually expanding the set. This method infers communication flow through PID relationships, providing a scalable and low-overhead solution for tracing Android's process-level interactions.

To monitor I/O events in Android, modern tracing features of the Linux kernel can be used that allow on-demand instrumentation of Linux kernel functions and memory address de-referencing. Consequently, it is possible to extract information from the structures of the kernel.

Tracing the vfs functions allows to access the suitable kernel structures. For example, in Unix-like kernels, vfs_ioctl is the VFS-layer entry point that dispatches device- or filesystem-specific control command.

```
long vfs_ioctl(struct file *filp,
               unsigned int cmd,
               unsigned long arg)
```

If we observe the struct file, we can see that it contains a pointer to a struct inode.

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /* cached value */
        const struct file_operations    *f_op;
```

inode, in turn, has some very useful fields, i_rdev, i_ino. r_dev is a 32-bit value that uniquely identifies a device node, containing the major and minor value, as defined in `/include/linux/kdev_t.h` [77]. Moreover, inode (i_ino) distinquishes a file inside a filesystem. Files in different filesystems can have the same inode. Thus, to uniquely identify regular files, non-device nodes, the st_dev inside struct `super_block *i_sb` is needed.

```
struct inode {
        umode_t                 i_mode;
        unsigned short          i_opflags;
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;
                        ...
        unsigned long           i_ino;
                        ...
        struct super_block      *i_sb;
                        ...
        dev_t                   i_rdev;
```

The `struct super_block` includes s_dev 32-bit value, combining the major and minor number. The major number selects which driver (or class of device) supplies the filesystem, while the minor chooses the specific instance of that driver (or which partition). s_dev and inode can uniquely identify a regular file (e.g. the sms database in the android context).

```
struct super_block {
        struct list_head        s_list;
        dev_t                   s_dev;
```

Finally, i_mode provides information about the permissions of a file and kuid and kgid fields refer to the owner and group of the file. That could be useful due to android sandboxing, meaning that all files owned by given process will have the same kuid.

Other than vfs functions tracing, IPC through binders, unix stream and datagram sockets are tracked.

At a conceptual level, Android's public APIs are self-documenting: the names of the methods an app invokes already encode the high-level actions being performed (for example, CameraManager.openCamera() clearly indicates camera use). By instrumenting the Android Framework and running a suite of representative applications, we can trace every API call down to its corresponding kernel activity. From these experiments a two-stage mapping was built: first, translation of observed kernel events back into specific Android API calls occurs; then each API call is associated with its semantic behavior (e.g., "use camera," "start audio recording," etc.). In this way, simply by monitoring low-level events, the high-level operations an application is executing can reliably be reconstructed.

To match specific behaviors like camera to the device node, Android's Java source files are parsed and two unique identifiers to each API function is assigned,one for entry and one for return. Then a write system call is added to each entry and return that is captured by vfs_write kprobe, using an argument equal to the API's uniqueID. Having that unique id and the r_dev, a device node is mapped with an API.

To sum up, the general idea is that the trace script with kprobes and tracepoints runs. Then, the output is used as input for the IPC slicing algorithms and finally these activities are mapped to specific android functionalities using a file that holds the devices r_dev. Given the description of the existing system, my contribution, recorded in the next chapters can be more clear.

The discussion above shows that SliceDroid provides the most readily generalisable behavioural-reconstruction pipeline among existing approaches. The next chapter explains how we extend this foundation to deliver plug-and-play behavioural profiling across devices.

Chapter 4

Methodology

4.1 Limitations of SliceDroid and Experimental Setup

The workflow for SliceDroid paper was tested on a Pixel9 device with Android 15 AOSP. However, porting instrumentation to a new device still requires some non-trivial setup work e.g. modifying a specific android framework version, compiling the modules changed and deploying to the device.

It seems that it is worth investigating ways to make the previously described behavioral profiling mechanism plug-and-play. So, to further analyze SliceDroid and extend it, my colleagues and i obtained 3 android devices, and, specifically Nothing phone 2a, OnePlus Nord CE4 Lite 5G, Samsung A15 and later Pixel9. The main selection criterion was the ease of the rooting procedure.

My primary focus began with the Nothing Phone 2a, which runs Android 13 with kernel version 5.15.104. Firstly, a meticulous rooting procedure was needed to enable the system-level modifications required for SliceDroid. Specifically, i followed the instruction for the specific android model found in xda-forum [78]. The process commenced with preparing the device: enabling Developer Options and subsequently activating OEM Unlocking within the system settings. This critical preliminary step allowed the device's bootloader to be unlocked, a prerequisite for flashing any custom or modified images. Unlocking the bootloader, while essential for our research, inherently carries risks, including a factory reset of the device and potential voiding of the manufacturer's warranty.

Following this, the Magisk tool, a popular systemless rooting solution, was selected due to its flexibility and broad community support, enabling modifications without directly altering the /system partition. The specific boot image for the Nothing Phone 2a, namely Pacman_U2.6-240828-1906 from the official Nothing OS firmware repository [79], was then acquired. This init_boot image, which contains critical boot-time components, was subsequently patched using the Magisk application. This patching process modifies the init_boot image to integrate Magisk's root capabilities in a systemless manner, allowing for root access and the installation of custom modules while maintaining system integrity.

With the bootloader unlocked, the device was rebooted into fastboot mode. The patched init_boot image was then flashed onto the device using the fastboot command-line tool. The command executed was `fastboot flash init_boot patched_magisk.img`, replacing

the original boot image with our Magisk-enabled version. Interestingly, unlocking the bootloader deletes user data to ensure privacy [80]. After unlocking, all data on the device is erased. Upon reboot, the successful installation of Magisk was verified by checking the Magisk application, confirming root access and readiness for SliceDroid's deployment.

4.2 Behavioral Reconstruction Portability Enhancement

The existing behavior-reconstruction workflow employed by SliceDroid is effective and intuitive; however, it has an inherent limitation. It must parse Android's Java source files in order to map low-level kernel events to high-level API calls. If that costly step could be avoided, it would be far easier to make SliceDroid run on a new device with only minimal setup. Because SliceDroid already relies on device nodes to detect high-level behavior, it is worth investigating whether those nodes can be discovered directly from Linux file-system entries such as `/dev`.

Regarding compatibility with devices other than Pixel, we preferred an approach that avoids "touching" AOSP framework files, because those edits would require re-injecting code into the Android framework for every single build, precisely the burden SliceDroid sought to remove.

Instead, the idea is as follows:

- Device-node identifiers (`r_dev`) can be retrieved directly from the shell.
- The high-level functionality that many device nodes offer can often be inferred from the owner, group or name of the file.

For example, on a Nothing 2a phone the command `find /dev -printf '%u\n'` sort | uniq produces:

```
audio, audioserver, bluetooth, camera, gps, logd, media, mtp, nfc,
    prng_seeder, radio, root, shell, statsd, system, uhid, wifi
```

The kernel's *uevent* mechanism further enriches the device-node context by emitting a small text file under directories such as `/vendor/etc` and `/system/etc/` whenever a device is registered or its state changes. That file contains ownership and permission information. By parsing *uevent* files directly on the device, one can derive both vendor-specified defaults and system-level overrides for owner (`kuid`) and group (`kgid`) assignments.

For each device node under `/dev` we recorded its name, owner, group and its raw device identifier (`r_dev`). The identifiers were obtained via the `stat` command, which returns both the major and minor device numbers. Specifically, using the -c flag to specify the format of the output, we can retrieve the major and minor number of a device node by using the `%T:%t` format sequence [81]. To represent each device or filesystem in a single 32-bit value we used the Linux kernel's `MKDEV` macro:

```
1 #define MINORBITS  20
2 #define MINORMASK  ((1U << MINORBITS) - 1)
3 #define MAJOR(dev) ((unsigned int)((dev) >> MINORBITS))
```

```
#define MINOR(dev) ((unsigned int)((dev) & MINORMASK))
#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

We then constructed a mapping of high-level behaviors to device nodes by matching each node's *owner*, *group*, or *name* to its corresponding functionality (camera, audio input, NFC, Bluetooth, GNSS). The logic for each category is detailed below:

- **NFC:** Any node whose owner, group, or filename contains the substring "nfc" is classified as an NFC device node.
- **GNSS:** Nodes are identified as GNSS if "gnss" or "gps" appears in the owner, group, or name.
- **Bluetooth:** Because Bluetooth device-node names are often non-descriptive, we require "bluetooth" to be present in either the owner or the group in order to classify the node as Bluetooth-related.
- **Audio Input:** We first search for the substring "pcm" in the device-node name. We then verify that the owner or group corresponds to the audio subsystem. PCM capture nodes follow the pattern

  $$\texttt{/dev/snd/pcmC}\,k\,\texttt{D}\,n\,\texttt{c},$$

  where k is the card number and n is the ADMAIF channel number minus one [82].
- **Camera:** All `video*` nodes are associated with camera behavior, since they represent "Video and metadata for capture/output devices," as noted in the kernel documentation [83]. Additionally, there exist vendor-specific customizations such as `lwis-*` device nodes, which are tailored to the Pixel camera HAL and hardware [84]. Detecting these custom nodes requires either prior knowledge of the device internals or a configuration file listing vendor-specific devices. For demonstration purposes, we include only the Pixel's LWIS nodes in our mapping.

The output is a JSON file whose keys are high-level behaviors and whose values are the associated `r_dev`. As a first validation, `/sys/kernel/tracing/tracing_on` was manually inspected to observe if using the camera actually lead to access in video* device nodes. Similar procces was followed for the other nodes. The complementary mapping for regular files and SQLite databases is discussed separately in Section 4.3.

Using this procedure, the creation of device-node mappings can be significantly streamlined. The most straightforward way to evaluate the effectiveness of the mechanism is by comparing the generated mapping file with the reference mapping produced through Android-Framework instrumentation, as presented in the original SliceDroid paper. Treating the paper's mapping as ground truth, the script was implemented and iteratively refined until it captured the relevant mappings on a Pixel 9 device. By using simple metrics such as *True Positives* (high-level functionalities correctly mapped in both the paper and the current implementation) and by analysing the causes of *False Positives* (cases where the name of a device node appears in the current implementation but not in the API-derived mapping), it becomes straightforward to identify both the strengths and limitations of the

proposed approach. At this point, it is important to note that the names of the device nodes are compares and not the r_dev because we cannot expect r_dev to be the same after updates, reboots etc. Generally, we can consider the names of the nodes to be stable, unless a significant driver update was introduced.

4.3 Databases and Network Monitoring

Tracing an application's interactions with on-device databases – contacts, SMS, call logs and calendar – reveals which sensitive records it handles locally. However, the network traffic a process generates can be equally important for security and privacy profiling. To address Research Question 2, this section unifies two evidence streams into a single, device-agnostic pipeline: (i) a map of character/block devices and SQLite data stores, and (ii) kernel-level probes that capture network activity.

4.3.1 Locating common SQLite databases

Database paths on most devices follow vendor-constrained templates For example:

```
/data/user*/com.android.providers.contacts/databases/contacts2.db
```

Therefore, a path-based search most of the time suffices to collect the `st_dev` and `inode` values needed for later matching.

One open question is whether parts of those databases could be cached solely in RAM via `mmap`. A review of Linux 5.15.104 (Nothing Phone 2a) showed that each `struct inode` points to a `struct address_space` describing both the disk file and its page cache, so inode-based matching remains valid [85].

To verify this in practice, we placed a `kprobe` on `vm_mmap_pgoff`. The probe showed that only auxiliary WAL files (for example, `contacts2.db-shm`) are memory-mapped, while the primary database file retains its original identifiers.

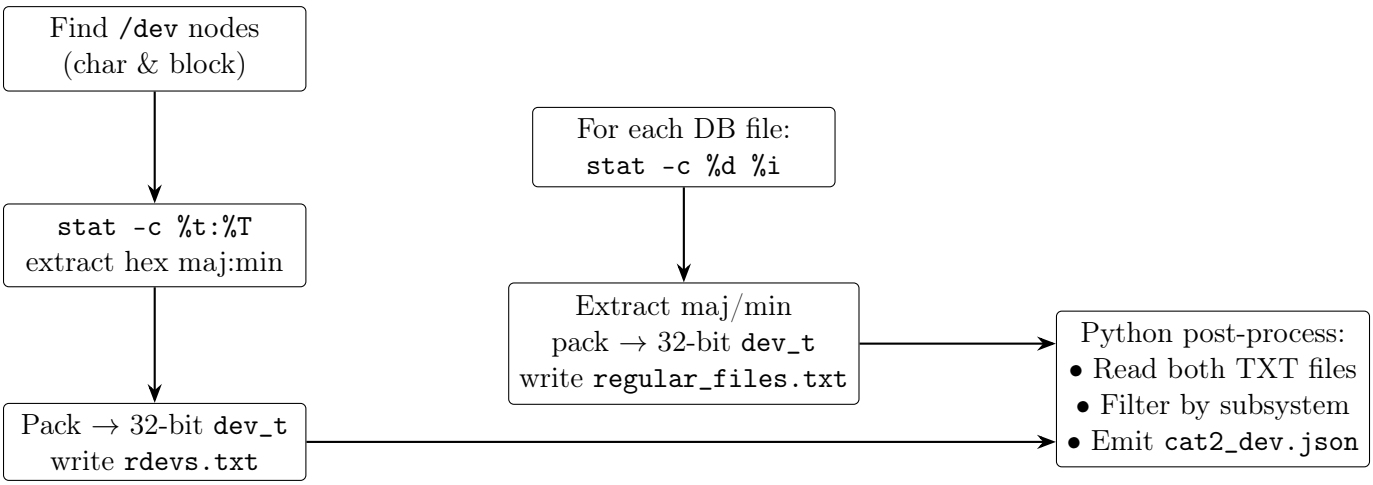


Figure 4.1: Workflow: device and file metadata extraction converging into a JSON mapping.

4.3.2 Network-level instrumentation

Building on the resource map, we extend the tracer with network-centric probes so that local resource accesses and external communications appear in a single event stream. Table 4.1 summarises the instrumented kernel functions and the rationale for each hook.

Kernel Function	**Reason for Instrumentation**
`tcp_sendmsg`	Logs data transmission over TCP, enabling tracking of sent payload size and destination.
`tcp_recvmsg`	Captures incoming TCP traffic, useful for measuring received data and connection responsiveness.
`udp_sendmsg`	Records UDP packet sends, important for analyzing stateless communications and their payload sizes.
`udp_recvmsg`	Tracks receipt of UDP packets, useful for profiling services using datagram protocols.

Table 4.1: Instrumented Kernel Functions for Network Activity Tracing

All captured events – device-node accesses, regular-file accesses and network operations – feed the same behavior-reconstruction layer, yielding a unified, timestamp-ordered trace that can be visualised in SliceDroid dashboard.

4.3.3 Demonstration APK and preliminary findings

As a proof-of-concept evaluation of Research Question 2, we developed an APK that the user can choose to open the events of calendar or sent an http request. When the APK runs, the tracing pipeline records both the database accesses and any network activity generated by the same process, letting the resulting events be explored in the SliceDroid user interface. A simple post-processing script then aggregates per-process traffic volumes (TCP versus UDP) and visualises the results, illustrating the utility of the unified pipeline and laying the groundwork for more sophisticated comparative analyses.

Chapter 5

Results

5.1 Device Nodes Mapping Results

As noted in Chapter 4.2, the effectiveness of our portability-oriented mapping mechanism can be quantified by comparing its output to the ground-truth device-node map published in the original SliceDroid study. Viewing the comparison as a binary classification task—each device node is either correctly mapped (*positive*) or not (*negative*)—yields the confusion-matrix terms:

- **TP** (true positives): device-node entries present in both the ground-truth map and our generated map.
- **FP** (false positives): entries produced by our implementation but absent from the ground truth.
- **FN** (false negatives): ground-truth entries that our script failed to capture.
- **TN** (true negatives): entries correctly omitted because they do not correspond to any high-level behaviour in SliceDroid. True negatives are not significant for our method.

From these counts we derive three standard metrics that succinctly summarise performance:

Recall measures coverage—*how many of the relevant special files did we capture?*

$$Recall = \frac{TP}{TP + FN}$$

Precision measures exactness—*of all the files we mapped, how many were actually correct?*

$$Precision = \frac{TP}{TP + FP}$$

False Negative Rate (FNR) indicates missed coverage—*how many relevant special files were overlooked by our method?*

$$FalseNegativeRate = \frac{FN}{TP + FN}$$

A high FNR suggests significant gaps in capturing relevant entries.

False Discovery Rate (FDR) measures incorrect mapping—*among files identified by our method, how many were incorrectly included?*

$$FalseDiscoveryRate = \frac{FP}{TP + FP}$$

A high FDR indicates that many identified entries do not correspond to actual special files relevant to SliceDroid.

High recall and low FNR indicate thorough coverage of the ground-truth entries. High precision and low FDR imply the method accurately identifies correct mappings while minimizing incorrect inclusions.

Upon evaluation, 28 True Positives, 31 False Positives and 0 False Negatives were found. That means that the method found all the significant nodes within our Pixel 9 ground truth set but includes more than double the original number of nodes. Thus, the results were the following:

$$Recall = \frac{28}{28} = 1$$

$$Precision = \frac{28}{28 + 31} = 0.47$$

$$FalseNegativeRate = \frac{0}{28 + 0} = 0$$

$$FalseDiscoveryRate = \frac{31}{31 + 28} = 0.53$$

Clearly, both the False Discovery Rate (FDR) and Precision remain non-satisfying, approximately 50%, indicating that our approach struggles with false positives and includes many device nodes absent from the ground truth. Distinguishing relevant from irrelevant nodes for each high-level behavior, however, is non-trivial.

During testing on a Pixel 9 device, we discovered several nodes used by core phone activities that the ground truth had omitted:

- `pcmC0D17c`, which captures the speaker's voice during a video call (e.g., via Meta's Messenger).
- `acd-chre_bt_offload_data_tx`, responsible for toggling Bluetooth on and off.
- `gnss_boot`, used to obtain the initial location fix when launching map services.

These examples likely do not exhaust all relevant cases; additional unlisted nodes may exhibit meaningful functionality. Consequently, our ground truth enumeration is not comprehensive. Were we to compare against an oracle with perfect behavior-to-node mappings, we would likely observe a lower FDR and higher Precision—and might even uncover Recall limitations of our method. Possible implications of the ground-truth generation algorithm, as well as our approach, are discussed in the *Threats to Validity* section.

5.2 Regular file access and network traffic integration Results

To evaluate regular-file accesses and network-traffic monitoring (**RQ2**), we developed a demonstration APK with the following capabilities, presented in 5.1.

Functionality	**Required permission**	**Android API / constant**
View calendar events	`android.permission.READ_CALENDAR`	`android.provider.CalendarContract`
HTTP GET request	`android.permission.INTERNET`	`java.net.HttpURLConnection`

Table 5.1: Capabilities exercised by the demo APK

With the APK installed, the user can select any of these actions to exercise and validate the tracing pipeline. Figure 5.1 shows the application's main menu:

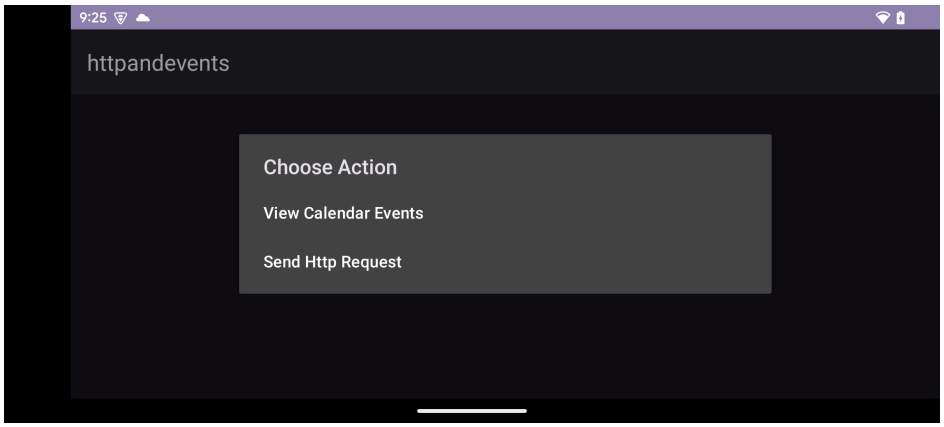
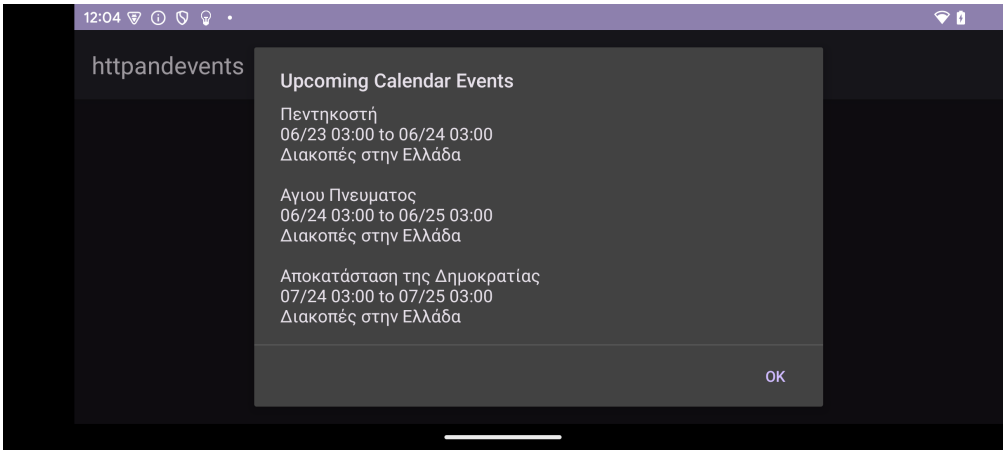


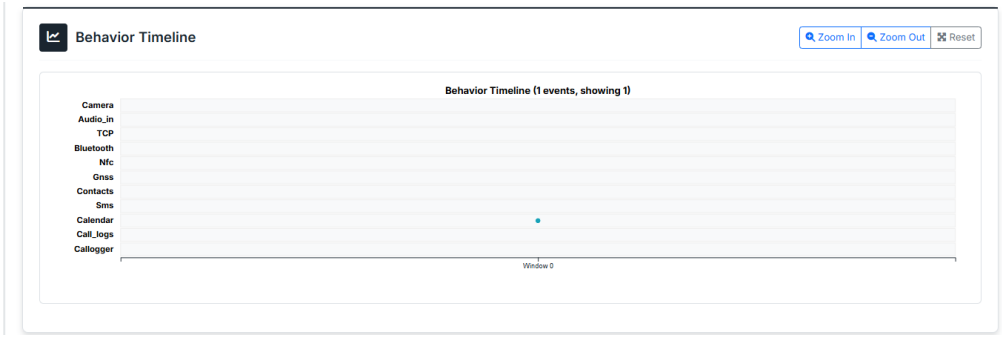
Figure 5.1: Demo APK main menu for selecting functionality

If we use the tracing script while using the apk and choose to see the calendar events, we can then observe the regular file access from the UI dashboard.

If we run the tracing script on the APK and enable "See calendar events," SliceDroid's UI dashboard immediately shows the corresponding file accesses (see Figure 5.2b). This demonstrates that even a process with elevated permissions—which might otherwise bypass Android framework restrictions—can be detected, allowing us to enumerate all accesses to the Calendar, SMS, and Call Log databases. By monitoring device-node operations at the virtual file system level, our low-level approach reliably captures every file interaction. Figure 5.2 presents an Android calendar access side by side with its dashboard trace.



(a) Calendar access from demo Apk



(b) Dashboard

Figure 5.2: Comparison of the calendar-view pop-up and the dashboard screen.

For completeness, we also perform a dummy HTTP request from the APK to gather protocol-usage statistics. Here, the app sends a request to `https://httpbin.org`; by filtering on the app's process ID, we capture the resulting network traffic with minimal modifications to our tracing infrastructure.

In the captured trace the APK ran under process ID 13124. After the dashboard filters were applied (Figure 5.3), minimal outbound TCP traffic was recorded (negligible, < 10 kB) while inbound TCP traffic totalled 0.16 MB. For UDP, 0.09 MB were received and none were transmitted. UDP traffic is attributed to the dns query that was needed to resolve the ip of the server. Figure 5.4 shows that, beyond packet sizes, the dashboard also counts the number of distinct network events. Asynchronous slicing was used, which allows threads spawned by the target process to be traced as well. The kernel probes tcp_sendmsg, tcp_recvmsg, udp_sendmsg, and udp_recvmsg were added to the slice configuration to capture these events.

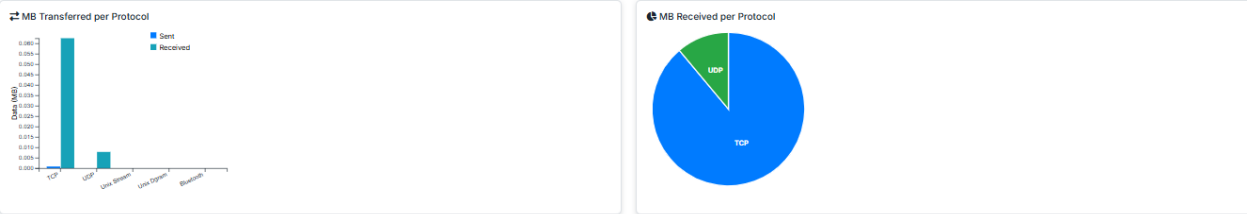


Figure 5.3: First visualization of network traffic through the dashboard

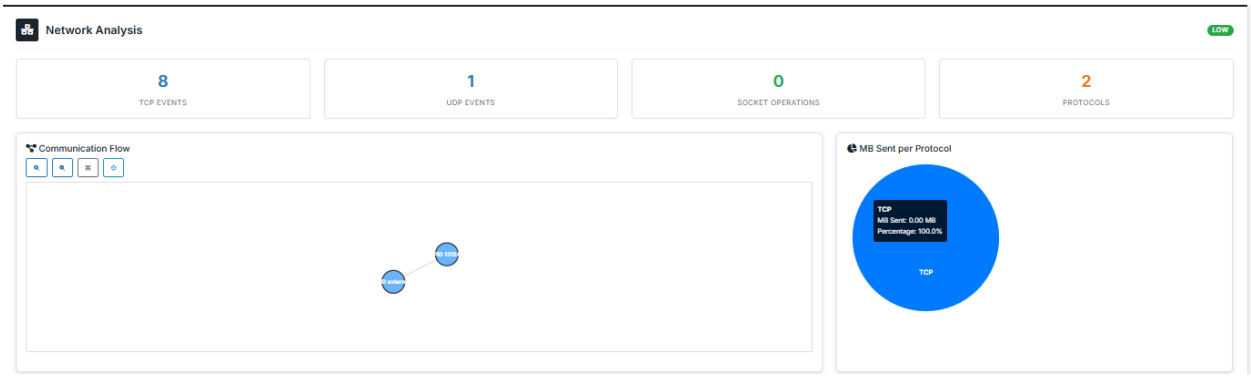


Figure 5.4: Second visualization of network traffic through the dashboard

5.3 Overall contribution

Figure 5.5 presents an overview of SliceDroid. Components outlined in **red** mark my individual contribution within this project.

The principal tasks I completed are summarised below:

- **Extended tracing script.** I added further kprobes and tracepoints and refactored the script for modularity, moving each `tracefs` path into a dedicated configuration file (e.g. `events_to_filter.txt`). These files appear inside the blue rectangle in Figure 5.5.
- **Resource-resolution layer.** For the resource resolution layer, i implemented a shell script to discover device nodes and database files on the target Android device. This logic is depicted in the upper-right area of the diagram.
- **Host-side processing.** On the host side, i developed a Python utility to map each discovered device node to its corresponding high-level functionality. This processing stage runs after the mobile-side instrumentation completes, enabling correct association between low-level traces and system resources.
- **Visualisation dashboard enhancements.** The visualization dashboard was also enhanced: i added trace-file preloading functionality and made minor user-interface improvements to streamline the analysis workflow.

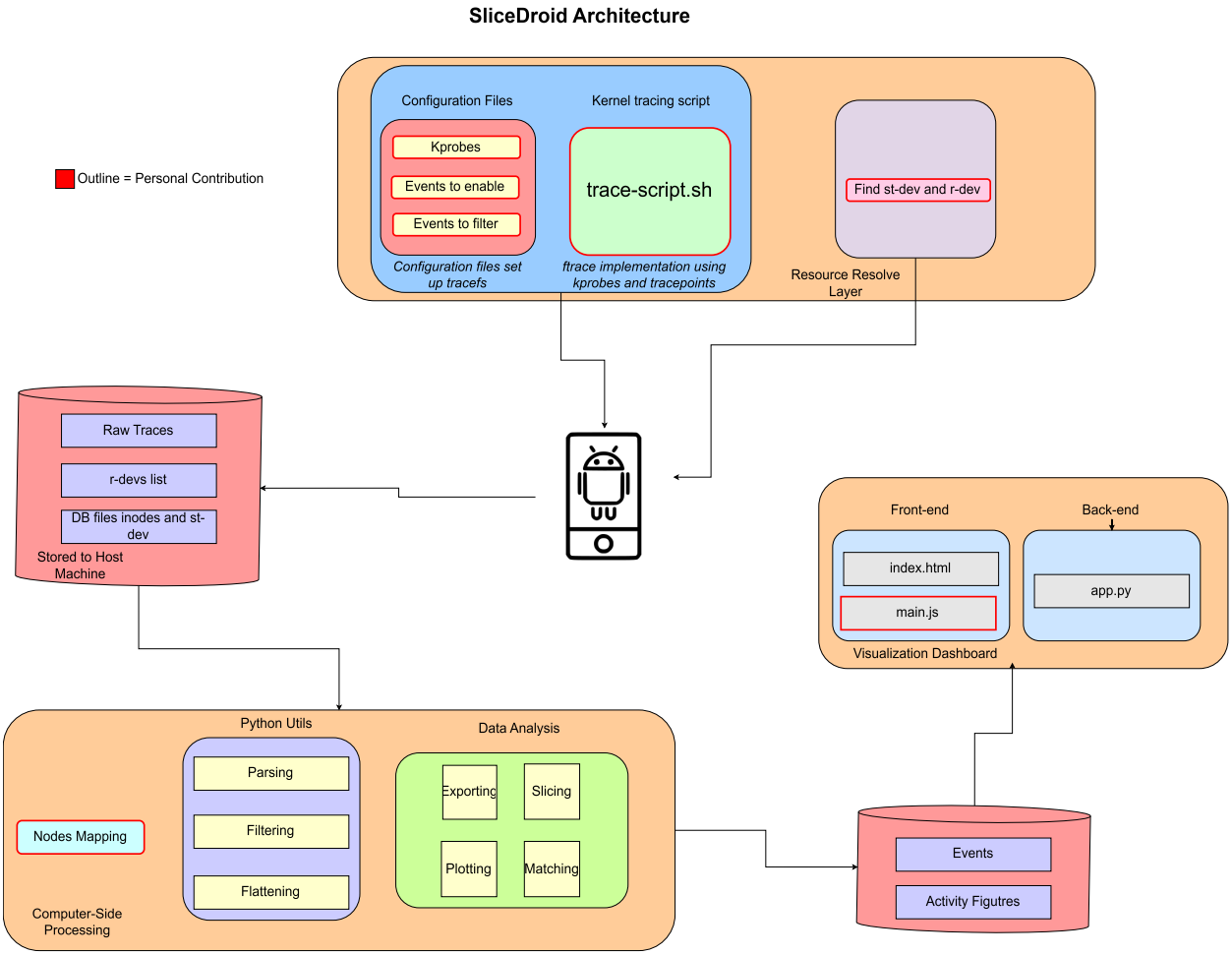


Figure 5.5: SliceDroid architecture; elements outlined in red are my personal contributions.

- **End-to-end orchestrator.** Finally, an orchestrator script was created to connect all components end-to-end. After installing the SliceDroid app on the device, the user can launch the entire pipeline and view the dashboard simply by running:

  ```
  python run_slicedroid.py
  ```

  This obviates manual steps such as pushing configuration files or pulling raw traces, thus improving usability and reducing manual steps.

Collectively, these enhancements lower the setup overhead and move SliceDroid closer to a plug-and-play behavioural-profiling tool for Android devices. The resulting platform performs dynamic analysis and reports the low-level activity of a selected APK. In practice, it helps users detect unusual behavioural patterns or simply understand how an executable exercises system resources.

During evaluation, for example, the system revealed that Messenger repeatedly accesses the contact list—behaviour that is legitimate given the permissions requested, yet still worth surfacing. The same workflow can be applied to any other Android application, enabling security analysts and developers to obtain quick, data-driven insights into an app's runtime behaviour without manual instrumentation.

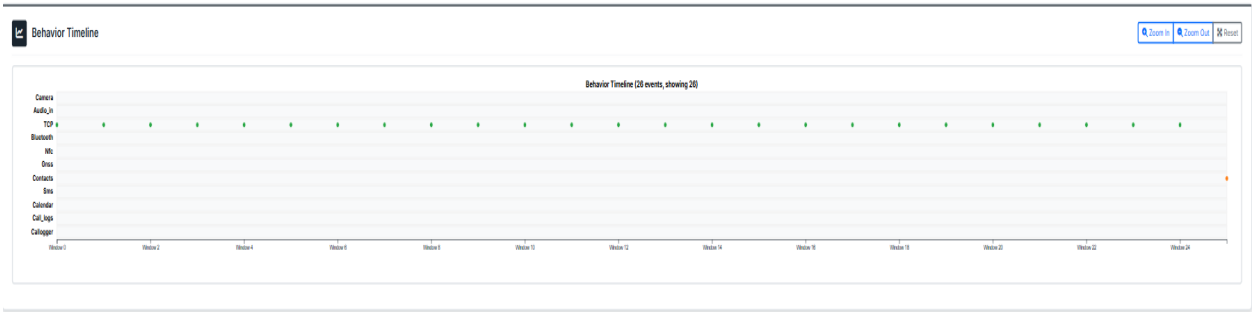


Figure 5.6: Meta's Messenger accessing the contacts database of Android (orange dot)

Chapter 6

Threats to Validity

First, our answer to Research Question 1 is based on explaining the extra device nodes we uncovered through manual trace inspection; we did not perform a formal validation step (e.g., by instrumenting the relevant Android APIs). Second, we conducted the evaluation on the same Pixel 9 handset used to build SliceDroid's reference set, meaning we already knew the "test" device nodes. That prior knowledge can bias the matching process—for example, it may have prompted us to recognise lwis-* nodes simply because they appeared during earlier trials. Consequently, the study should be viewed as an illustrative proof-of-concept rather than an exhaustive assessment; the limited experimental scale reflects this intention.

Moreover, the discussed approach relies on kernel frameworks like `v4l2` and the built-in device manager of android, `ueventd`. The V4L2 drivers tend to be very complex and can export multiple device nodes [86]. After manually inspecting the camera and trying all the functionalities some device nodes could not be accessed (i.e video8-video11). So, the probability of some False Positive nodes being unused by the system is considered. Next, Android's core defines the set of system users and groups—including nfc, camera, bluetooth, gps, etc.—in the AOSP header which maps each name to a fixed UID/GID at build time [87]. At boot, the ueventd daemon reads from /vendor/etc/ueventd.rc and /system/etc/uevent.rc and applies those predefined owners, groups, and permissions to each /dev/* node it creates. So, a vendor miss-labeling or adding an unexpected label for owner or group could affect the approach.

The approach inherits the limitations of SliceDroid about the construction of the ground truth. SliceDroid uses an algorithm that matches device Nodes to API methods. For each device node, API methods that appear often in the same window as the device node but rarely in a window without the device node were identified. Rarely and often are notated through the algorithm's hyperparameters α and β. These hyperparameters are set empirically, thus meaning that some device nodes could be falsely included or not included. Moreover, regarding the instrumented java APIs used, probably some packages regarding audio, camera etc. were omitted, meaning that they were not provided as input for the match APIs to device nodes algorithm.

Chapter 7

Conclusion

This thesis systematically addressed the portability challenges associated with kernel-level tracing in Android devices, with a specific focus on behavioral reconstruction using the SliceDroid approach. The investigation highlighted significant issues arising from Android's fragmentation, vendor customizations, and restrictive kernel configurations that limit kernel-level tracing tools such as ftrace and kprobes.

The empirical evaluation conducted across multiple Android devices, including Pixel 9, Nothing Phone 2a, OnePlus Nord CE4 Lite 5G, and Samsung A15, demonstrated the feasibility of creating a device-agnostic tracing pipeline. The results showed that kernel-level tracing is viable for capturing high-level behavioral data, but requires careful consideration of device-specific nuances such as kernel symbol visibility and vendor-specific customizations.

Using SliceDroid as a baseline, this study introduced methods to streamline the identification and mapping of device nodes to Android functionalities without modifying the Android Framework itself. Additionally, an enhanced tracing mechanism was developed to include regular files and database access tracing, complemented by integrated network monitoring, providing a more holistic view of Android apps high-level behavior.

Bibliography

[1] "Operating System Market Share Worldwide," https://gs.statcounter.com/os-market-share#monthly-202502-202502-bar, 2025, accessed: 2025-04-04.

[2] Android Open Source Project, "Kernel overview," https://source.android.com/docs/core/architecture/kernel, 2025.

[3] Steven Rostedt, "ftrace - Function Tracer," https://www.kernel.org/doc/html/v4.17/trace/ftrace.html, 2017, accessed: 2025-04-04.

[4] Jim Keniston and Prasanna S Panchamukhi and Masami Hiramatsu, "Kernel Probes (Kprobes)," https://docs.kernel.org/trace/kprobes.html, accessed: 2025-04-04.

[5] "The LTTng Documentation," https://lttng.org/docs/v2.13/, 2023, accessed: 2025-04-04.

[6] X. Zhang, A. Mathur, L. Zhao, S. Rahmat, Q. Niyaz, A. Javaid, and X. Yang, "An Early Detection of Android Malware Using System Calls based Machine Learning Model," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ser. ARES '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3538969.3544413

[7] H. Liu, D. J. Leith, and P. Patras, "Android OS Privacy Under the Loupe – A Tale from the East," in *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 31–42. [Online]. Available: https://doi.org/10.1145/3558482.3581775

[8] V. Gohil, N. Ujjainkar, J. Mekie, and M. Awasthi, "Performance optimization opportunities in the android software stack," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 1, no. 1, p. 100003, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S277248592100003X

[9] L. Gelle, N. Ezzati-Jivan, and M. R. Dagenais, "Combining distributed and kernel tracing for performance analysis of cloud applications," *Electronics*, vol. 10, no. 21, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/21/2610

[10] M. Desnoyers and M. R. Dagenais, "Tracing for hardware, driver and binary reverse engineering in linux," in *Ottawa Linux Symposium (OLS)*, 2006, available at: http://ltt.polymtl.ca/documentations.html. [Online]. Available: http://ltt.polymtl.ca

[11] D. Geer, B. Tozer, and J. S. Meyers, "For good measure: Counting broken links: A quant's view of software supply chain security," *USENIX ;login:*, vol. 45, no. 4, pp. 84–86, Winter 2020. [Online]. Available: https://www.usenix.org/publications/login/winter2020/geer

[12] X. Wang, Y. Zhang, X. Wang, Y. Jia, and L. Xing, "Union under duress: Understanding hazards of duplicate resource mismediation in android software supply chain," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3403–3420. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wang-xueqiang-duress

[13] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[14] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 317–326. [Online]. Available: https://doi.org/10.1145/2133601.2133640

[15] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[16] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. USA: USENIX Association, 2012, p. 29.

[17] K. Tam, S. J. Khan, A. Fattoriy, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, Feb. 2015, pp. 1–15, conference: NDSS Symposium, February 8–11, 2015.

[18] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu, "Using provenance patterns to vet sensitive behaviors in android apps," in *Security and Privacy in Communication Networks*, B. Thuraisingham, X. Wang, and V. Yegneswaran, Eds. Cham: Springer International Publishing, 2015, pp. 58–77.

[19] N. Alexopoulos, S. Althaus, and D. Spinellis, "Slicedroid: Towards reconstructing android application i/o behaviors from kernel traces," Jul. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15784277

[20] Android Open Source Project, "Understand systrace," 2 2025, last updated 2025-02-27 UTC; accessed 2025-05-20. [Online]. Available: https://source.android.com/docs/core/tests/debug/systrace

[21] Perfetto Open Source Project, "Tracing 101," 2025, accessed 2025-05-20. [Online]. Available: https://perfetto.dev/docs/tracing-101

[22] Android Open Source Project, "Generic Kernel Image (GKI) project," https://source.android.com/docs/core/architecture/kernel/generic-kernel-image, 2025, accessed: 2025-04-04.

[23] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 15–24.

[24] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 987–992. [Online]. Available: https://doi.org/10.1145/2950290.2983958

[25] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 226–237. [Online]. Available: https://doi.org/10.1145/2970276.2970312

[26] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 409–423.

[27] L. Nguyen-Vu, J. Ahn, and S. Jung, "Android fragmentation in malware detection," *Computers and Security*, vol. 87, p. 101573, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404819301361

[28] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2619091

[29] D. Nisi, A. Bianchi, and Y. Fratantonio, "Exploring {Syscall-Based} semantics reconstruction of android applications," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 517–531.

[30] Android Open Source Project, "Privacy indicators," 4 2025, last updated 2025-04-04 UTC; accessed 2025-05-20. [Online]. Available: https://source.android.com/docs/core/permissions/privacy-indicators

[31] National Vulnerability Database, "Cve-2023-21083 detail," 4 2023, last modified 2025-02-05; accessed 2025-05-20. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-21083

[32] Android Open Source Project, "Architecture overview," https://source.android.com/docs/core/architecture#architecture, 2025, accessed: 2025-05-04.

[33] M. Kamran, J. Rashid, and M. Nisar, "Android fragmentation classification, causes, problems and solutions," *International Journal of Computer Network and Information Security*, vol. 14, pp. 992–999, 09 2016.

[34] K. Yaghmour and M. Dagenais, "The linux trace toolkit," *Linux Journal*, no. 73, May 2000. [Online]. Available: https://www.linuxjournal.com/article/3829

[35] "Linux trace toolkit reference manual," Opersys Inc., 2004, accessed: 2025-04-06. [Online]. Available: https://www.opersys.com/LTT/dox/ltt-online-help/ltt-introduction.html

[36] M. Desnoyers and M. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," *OLS (Ottawa Linux Symposium)*, 01 2006.

[37] D. Goulet, "Unified kernel/user-space efficient linux tracing architecture," Master's thesis, École Polytechnique de Montréal, April 2012, accessed: 2025-04-06. [Online]. Available: https://publications.polymtl.ca/842/1/2012_DavidGoulet.pdf

[38] Kernel Newbies, "Linux 2.6.27 changelog," 2008, accessed: 2025-05-20. [Online]. Available: https://kernelnewbies.org/Linux_2_6_27#head-27c23aa79ee6da975ef95b4fff381ace1667a264

[39] ——, "Linux 2.6.28 changelog," 2008, accessed: 2025-05-20. [Online]. Available: https://kernelnewbies.org/Linux_2_6_28

[40] J. Corbet, "What's coming in 2.6.27," *LWN.net*, 2008, accessed: 2025-06-10. [Online]. Available: https://lwn.net/Articles/291091/

[41] S. Rostedt, "ftrace - function tracer," The Linux Kernel Documentation, 2008, accessed: 2025-04-06. [Online]. Available: https://docs.kernel.org/trace/ftrace.html

[42] ——, "[PATCH 00/16 v3] tracing: Add new file system tracefs," Linux Kernel Mailing List, Jan. 2015, mon, 26 Jan 2015 10:09:13 -0500. Available at https://lkml.org/lkml/2015/1/26/454.

[43] T. Bird, "Measuring function duration with ftrace," *Proceedings of the Linux Symposium*, 01 2009.

[44] A. R. Ghods, "A study of linux perf and slab allocation sub-systems," Master's thesis, University of Waterloo, January 2016, accessed: 2025-04-06. [Online]. Available: http://hdl.handle.net/10012/10184

[45] P. W. Contributors, "perf: Linux profiling with performance counters," 2024, accessed: 2025-04-06. [Online]. Available: https://perfwiki.github.io/main/

[46] M. Gebai and M. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys*, vol. 51, pp. 1–33, 03 2018.

[47] E. Rocca, "A brief introduction to systemtap," October 2019, accessed: 2025-04-06. [Online]. Available: https://www.linux.it/~ema/posts/systemtap-intro/

[48] Red Hat, Inc., *SystemTap Beginner's Guide*, Red Hat, 2007, accessed: 2025-04-06. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/systemtap_beginners_guide/scripts#scripts

[49] ——, *SystemTap Beginner's Guide: Understanding How SystemTap Works*, Red Hat, 2007, accessed: 2025-04-06. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/systemtap_beginners_guide/understanding-how-systemtap-works#understanding-architecture-tools

[50] ——, *SystemTap Beginner's Guide: SystemTap vs. Other Tools*, Red Hat, 2007, accessed: 2025-04-06. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/systemtap_beginners_guide/intro-systemtap-vs-others#intro-systemtap-vs-others

[51] eBPF Foundation. (2023) What is ebpf? Accessed: 2025-04-06. [Online]. Available: https://ebpf.io/what-is-ebpf/

[52] eBPF.io, "What is ebpf? — jit compilation," 2024, accessed: 2025-05-22. [Online]. Available: https://ebpf.io/what-is-ebpf/#jit-compilation

[53] eBPF Community, "eBPF: Introduction, Tutorials & Community Resources," 2025, accessed: 2025-04-06. [Online]. Available: https://ebpf.io/

[54] Android Developers. (2024) Overview of system tracing and perfetto. Accessed: 2025-04-06. [Online]. Available: https://developer.android.com/topic/performance/tracing

[55] Perfetto Project, "Atrace: Android system and app trace events," accessed: 2025-05-25. [Online]. Available: https://perfetto.dev/docs/data-sources/atrace

[56] ——, "Heap profiler," 2024, accessed: 2025-05-15. [Online]. Available: https://perfetto.dev/docs/data-sources/native-heap-profiler

[57] ——, "Perfetto github releases," 2025, accessed: 2025-04-06. [Online]. Available: https://github.com/google/perfetto/releases

[58] Perfetto Open Source Project, "System calls," 2025, accessed 2025-05-20. [Online]. Available: https://perfetto.dev/docs/data-sources/syscalls

[59] M. Chen, J. Keniston, K. Kressin, H. Patil, J. Pereira, and M. Probert, "Locating system problems using dynamic instrumentation," in *Proceedings of the Linux Symposium*, 2005, pp. 57–72, accessed: 2025-04-06. [Online]. Available: https://www.kernel.org/doc/ols/2005/ols2005v2-pages-57-72.pdf

[60] eBPF Documentation Contributors, "Trampolines," 2025, accessed: 2025-04-06. [Online]. Available: https://docs.ebpf.io/linux/concepts/trampolines/

[61] L. Rice, *Learning eBPF: Programming the Linux Kernel Without Writing Kernel Code*. Isovalent, 2022, accessed: 2025-04-06. [Online]. Available: https://cilium.isovalent.com/hubfs/Learning-eBPF%20-%20Full%20book.pdf?utm_source=chatgpt.com

[62] B. Gregg, "Linux perf examples," 2020, accessed: 2025-04-06. [Online]. Available: https://www.brendangregg.com/perf.html

[63] PerfWiki Contributors, "Linux kernel profiling with perf," 2024, accessed: 2025-05-25. [Online]. Available: https://perfwiki.github.io/main/tutorial/

[64] Eunomia Community, "Categorization of eBPF Hooks and Use Cases," 2025, accessed: 2025-04-06. [Online]. Available: https://eunomia.dev/others/miscellaneous/ebpf-usecases/

[65] eBPF Documentation Contributors, "Verifier," 2024, accessed: 2025-04-06. [Online]. Available: https://docs.ebpf.io/linux/concepts/verifier/

[66] Android Open Source Project, "Use DebugFS in Android 12," https://source.android.com/docs/core/architecture/kernel/using-debugfs-12, 2025, accessed: 2025-04-04.

[67] ——, "Running perfetto," accessed: 2025-05-25. [Online]. Available: https://android.googlesource.com/platform/external/perfetto/+/refs/heads/pie-gsi/docs/running.md

[68] "SELinux States and Modes," Red Hat Customer Content Services, 2014, accessed: 2025-06-16. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/sect-security-enhanced_linux-introduction-selinux_modes

[69] Linus Torvalds and Dan Rosenberg, "kptr_restrict for hiding kernel pointers from unprivileged users - kernel/git/torvalds/linux.git - Linux kernel source tree," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=455cd5ab305c90ffc422dd2e0fb634730942b257, 2011, accessed: 2025-04-04.

[70] B. Gregg, "Linux bcc/bpf tcplife: Tcp lifespans," https://www.brendangregg.com/blog/2016-11-30/linux-bcc-tcplife.html, 2016, accessed: 2025-04-04.

[71] J. Corbet, "The state of system observability with bpf," https://lwn.net/Articles/787131/, 2019, accessed: 2025-04-04.

[72] A. O. S. Project, "Develop kernel code for gki," https://source.android.com/docs/core/architecture/kernel/kernel-code, 2025, accessed: 2025-04-04.

[73] P. Zijlstra, "sched: Change task_struct::state," https://github.com/torvalds/linux/commit/2f064a59a11ff9bc22e52e9678bc601404c7cb34, 2021, accessed: 2025-04-07.

[74] J. Marchand, "Tools broken by the renaming of the state field of task_struct," https://github.com/iovisor/bcc/issues/3658, 2021, accessed: 2025-04-07.

[75] eBPF Docs, "BTF," https://docs.ebpf.io/concepts/btf/, accessed: 2025-06-23.

[76] eBPF Documentation Contributors, "BPF CO-RE (Compile Once - Run Everywhere)," https://docs.ebpf.io/concepts/core/, accessed: 2025-04-07.

[77] "kdev_t.h (line 8)," Bootlin Elixir, 2025, accessed: 2025-05-15. [Online]. Available: https://elixir.bootlin.com/linux/v5.15.104/source/include/linux/kdev_t.h#L8

[78] XDA Developers Forum, "Guide: Bootloader Unlock and Rooting with Magisk," 2025, accessed: 2025-04-15. [Online]. Available: https://xdaforums.com/t/guide-bootloader-unlock-and-rooting-with-magisk.4662722/

[79] spike0en, "nothing_archive: Ii. unlocking bootloader," 2025, accessed: 2025-05-15. [Online]. Available: https://github.com/spike0en/nothing_archive?tab=readme-ov-file#ii-unlocking-bootloader-

[80] Android Open Source Project, "Running tests," 2025, accessed: 2025-06-10. [Online]. Available: https://source.android.com/docs/setup/test/running

[81] M. Meskes, *stat: display file or file system status*, Free Software Foundation, January 2025, license GPLv3+; accessed 2025-06-23. [Online]. Available: https://man7.org/linux/man-pages/man1/stat.1.html

[82] NVIDIA Corporation, *Audio Setup and Development*, NVIDIA Corporation, Online, Feb. 2025, last updated on February 25, 2025. [Online]. Available: https://docs.nvidia.com/jetson/archives/r36.4.3/DeveloperGuide/SD/Communications/AudioSetupAndDevelopment.html

[83] The Kernel Development Community, *1.1. Opening and Closing Devices*, The Linux Kernel Documentation, Jun. 2025, accessed: 2025-06-18. [Online]. Available: https://docs.kernel.org/userspace-api/media/v4l/open.html

[84] The Android Open Source Project, "LWIS: Lightweight Image Subsystem Driver," https://android.googlesource.com/kernel/google-modules/lwis/, 2025, accessed: 2025-06-18.

[85] The Linux Kernel Archive, "Linux kernel source: `include/linux/fs.h`," https://elixir.bootlin.com/linux/v5.15.104/source/include/linux/fs.h#L438, 2025, accessed: 2025-05-17, cited line 438.

[86] J. Corbet, "Documentation/video4linux/v4l2-framework.txt," LWN.net, Jan. 2009, accessed: 2025-06-18. [Online]. Available: https://lwn.net/Articles/313784/

[87] Android Open Source Project, "android_filesystem_config.h," https://android.googlesource.com/platform/system/core/+/android-7.1.2_r33/include/private/android_filesystem_config.h, 2017, accessed: 2025-06-18.

Appendix

The following shell script is utilised for configuring and executing kernel-level tracing via kprobes on Android devices:

```
#!/system/bin/sh

CONFIG_DIR="/data/local/tmp/config_files"
TRACE_DIR="/sys/kernel/tracing"
TMP_DIR="/data/local/tmp"

rm -f $TMP_DIR/trace.trace
rm -f $TMP_DIR/trace.trace.gz

# Configure tracing options
echo 102400 > $TRACE_DIR/buffer_size_kb
echo record-tgid > $TRACE_DIR/trace_options

# Disable all events and clear old data
echo 0 > $TRACE_DIR/tracing_on
echo 0 > $TRACE_DIR/events/enable
echo > $TRACE_DIR/trace
echo > $TRACE_DIR/kprobe_events
echo > $TRACE_DIR/events/binder/filter

# Load basic kprobes
if [ -f "$CONFIG_DIR/kprobes.txt" ]; then
    while read -r probe; do
        echo "$probe" >> $TRACE_DIR/kprobe_events
    done < "$CONFIG_DIR/kprobes.txt"
fi

# Load conditional kprobes by device tag
device_tag="generic"
if [ "$(getprop ro.product.brand)" = "google" ]; then
    device_tag="pixel"
fi

if [ -f "$CONFIG_DIR/kprobes_conditional.txt" ]; then
    while IFS='->' read -r tag probe; do
```

```
        [ "$tag" = "$device_tag" ] && echo "$probe" >> $TRACE_DIR/
           kprobe_events
    done < "$CONFIG_DIR/kprobes_conditional.txt"
fi

# Collect PIDs
all_pids=""
if [ -f "$CONFIG_DIR/pid_targets.txt" ]; then
    while IFS=',' read -r mode name; do
        if [ "$mode" = "-x" ]; then
            pids=$(pgrep -x "$name")
        else
            pids=$(pgrep "$mode")
        fi
        all_pids="$all_pids $pids"
    done < "$CONFIG_DIR/pid_targets.txt"
fi

# Create filter string
pid_string=""
for pid in $all_pids; do
    [ -n "$pid_string" ] && pid_string="$pid_string && "
    pid_string="${pid_string}common_pid != $pid"
done

# Apply event filters
if [ -f "$CONFIG_DIR/events_to_filter.txt" ]; then
    while read -r event; do
        if [ -d "$TRACE_DIR/$event" ] && [ -w "$TRACE_DIR/$event/
           filter" ]; then
            echo "($pid_string)" > $TRACE_DIR/$event/filter 2>/dev/
               null
        fi
    done < "$CONFIG_DIR/events_to_filter.txt"
fi

# Enable selected events
if [ -f "$CONFIG_DIR/events_to_enable.txt" ]; then
    while read -r event; do
        if [ -d "$TRACE_DIR/$event" ] && [ -w "$TRACE_DIR/$event/
           enable" ]; then
            echo 1 > "$TRACE_DIR/$event/enable" 2>/dev/null
        fi
    done < "$CONFIG_DIR/events_to_enable.txt"
fi

# Enable vendor-specific events
```

```
79 if [ -f "$CONFIG_DIR/events_to_enable_conditional.txt" ]; then
80     while IFS='->' read -r tag event; do
81         if [ "$tag" = "$device_tag" ] && [ -d "$TRACE_DIR/$event" ]
               && [ -w "$TRACE_DIR/$event/enable" ]; then
82             echo 1 > "$TRACE_DIR/$event/enable" 2>/dev/null
83         fi
84     done < "$CONFIG_DIR/events_to_enable_conditional.txt"
85 fi
86
87 # IMPORTANT: Enable tracing AFTER all configuration is done
88 echo 1 > $TRACE_DIR/tracing_on
89 echo "Tracing enabled. Starting data collection..."
90
91 # Start trace_pipe background read
92 cat $TRACE_DIR/trace_pipe > $TMP_DIR/trace.trace &
93 waitpid=$!
94
95 echo "Trace collection active. Press ENTER to stop..."
96
97 # Wait for user to stop
98 read STOP
99
100 echo "Stopping trace collection..."
101
102 # Stop tracing
103 echo 0 > $TRACE_DIR/tracing_on
104
105 # Background sleep for timeout
106 sleep 10 &
107 timeout_pid=$!
108
109 # Wait for either process to finish
110 wait -n $waitpid $timeout_pid 2>/dev/null
111
112 if kill -0 $waitpid 2>/dev/null; then
113     echo "Timeout expired. Killing trace_pipe (pid $waitpid)."
114     kill $waitpid
115 else
116     echo "Trace collection completed before timeout."
117 fi
118
119 # Show output trace file
120 echo "Trace file info:"
121 ls -alh $TMP_DIR/trace.trace
122
123 # Gzip the trace
124 echo "Starting gzip compression..."
```

```
gzip -f $TMP_DIR/trace.trace
echo "Compression completed."

# Show gzipped file
echo "Final compressed trace file:"
ls -alh $TMP_DIR/trace.trace.gz
```

Listing 1: Kernel-level tracing script

Script for r_dev and st_dev,inode.

```
#!/bin/sh

# Constants from <linux/kdev_t.h>
MINORBITS=20
MINORMASK=$(( (1 << MINORBITS) - 1 ))


OUT_DEV="/data/local/tmp/rdevs.txt"
OUT_FILES="/data/local/tmp/regular_files.txt"

# Initialize output files
: >"$OUT_DEV"
: >"$OUT_FILES"

# Process all character/block devices under /dev
find /dev -mindepth 1 \( -type c -o -type b \) -print0 |
while IFS= read -r -d '' path; do
    name=${path#/dev/}

    # get original hex major:minor
    hex=$(stat -c '%t:%T' "$path")
    owner=$(stat -c '%U' "$path")
    group_owner=$(stat -c '%G' "$path")
    maj_hex=${hex%%:*}
    min_hex=${hex##*:}

    maj=$((0x$maj_hex))
    mino=$((0x$min_hex))

    rdev32=$(( (maj << MINORBITS) | (mino & MINORMASK) ))
    printf '%s %d %s %s\n' "$name" "$rdev32" "$owner" "$group_owner"
done |
sort > "$OUT_DEV"

# Select database paths depending on manufacturer
manufacturer=$(getprop ro.product.manufacturer)
if [ "$manufacturer" = "Samsung" ]; then
    db_paths=(
```

```
        "/data/data/com.samsung.android.providers.contacts/databases/
           contacts2.db"
        "/data/data/com.samsung.android.providers.telephony/databases
           /mmssms.db"
        "/data/data/com.samsung.android.providers.calendar/databases/
           calendar.db"
        "/data/data/com.samsung.android.providers.contacts/databases/
           calllog.db"
    )
else
    db_paths=(
        "/data/data/com.android.providers.contacts/databases/
           contacts2.db"
        "/data/data/com.android.providers.telephony/databases/mmssms.
           db"
        "/data/data/com.android.providers.calendar/databases/calendar
           .db"
        "/data/data/com.android.providers.contacts/databases/calllog.
           db"
    )
fi

for file in "${db_paths[@]}"; do

        # 1) get inode
        ino=$(stat -c '%i' "$file")

        # 2) get full major:minor in hex, use %d:%D because
        #they are not character/block device special files
        hex=$(stat -c '%d:%D' "$file")

        # 3) split into hex-major and hex-minor
        maj_hex=${hex%%:*}
        min_hex=${hex##*:}
        maj=$((0x$maj_hex))
        mino=$((0x$min_hex))

        # 4) pack into 32-bit dev_t
        dev32=$(( (maj << MINORBITS) | (mino & MINORMASK) ))

        # 5) emit file, dev32, inode
        printf '%s %d %s\n' "$file" "$dev32" "$ino"

done |
sort >"$OUT_FILES"

echo "Wrote device mapping to $OUT_DEV"
```

```
78 echo "Wrote file mapping   to $OUT_FILES"
```

Listing 2: Find the r_dev of all devices nodes in /dev and st_dev-inode for regular files

```
1  from collections import defaultdict
2  import json
3  import os
4  import sys
5  # Path to the mappings directory where the JSON file will be saved
6  MAPPINGS_DIR = os.path.join("data", "mappings")
7
8  # Path to the data directory where the sh output files are located
9  DATA_DIR = os.path.join("data", "nodes_and_files_data")
10
11 #check if the directories exists, if not create them
12 if not os.path.exists(MAPPINGS_DIR):
13     os.makedirs(MAPPINGS_DIR)
14 if not os.path.exists(DATA_DIR):
15     os.makedirs(DATA_DIR)
16
17 output_txt = os.path.join(MAPPINGS_DIR, 'cat2devs.txt')
18
19 # Use correct paths to the files
20 rdevs_path = os.path.join(DATA_DIR, 'rdevs.txt')
21 regular_files_path = os.path.join(DATA_DIR, 'regular_files.txt')
22
23 device_nodes = defaultdict(list)
24
25 # Uses r_dev to identify the device node
26 with open(rdevs_path, 'r') as f:
27     lines = f.readlines()
28
29 for line in lines:
30     if 'camera'== line.split(' ')[2] or 'camera'== line.split(' ')
       [3].removesuffix('\n'):
31         device_nodes['camera'].append(line.split(' ')[1])
32     if 'pcm' in line and line.split(' ')[0].endswith('c') and \
33         ('audio'== line.split(' ')[3].removesuffix('\n') or 'audio'==
              line.split(' ')[2]):
34         device_nodes['audio_in'].append(line.split(' ')[1])
35     if 'nfc'== line.split(' ')[2] or 'nfc'== line.split(' ')[3].
       removesuffix('\n'):
36         device_nodes['nfc'].append(line.split(' ')[1])
37     if 'gps'== line.split(' ')[2] or 'gps'== line.split(' ')[3].
       removesuffix('\n'):
38         device_nodes['gnss'].append(line.split(' ')[1])
39     if 'bluetooth'== line.split(' ')[2] or 'bluetooth'== line.split('
         ')[3].removesuffix('\n'):
```

```
        device_nodes['bluetooth'].append(line.split(' ')[1])

# Uses st_dev and i_node to identify the file
with open(regular_files_path, 'r') as f:
    lines = f.readlines()

for line in lines:
    if 'calllog' in line:
        device_nodes['callogger'].append(f"{line.split(' ')[1]} - {
           line.split(' ')[2]}".replace('\n',''))
    if 'contacts' in line:
        device_nodes['contacts'].append(f"{line.split(' ')[1]} - {
           line.split(' ')[2]}".replace('\n',''))
    if 'sms' in line:
        device_nodes['sms'].append(f"{line.split(' ')[1]} - {line.
           split(' ')[2]}".replace('\n',''))
    if 'calendar' in line:
        device_nodes['calendar'].append(f"{line.split(' ')[1]} - {
           line.split(' ')[2]}".replace('\n',''))

# Save the output to the mappings directory
with open(output_txt, 'w') as f:
    f.write(json.dumps(device_nodes, indent=4))

print(f"TXT file created at: {output_txt}")
```

Listing 3: Map the device node to high-level android functionality

Below the comparison between the ground truth json and the generated json is provided:

Thesis Approach Nodes


```
{
  "bluetooth": {
    "514850827": ["acd-chre_bt_offload_ctl"],
    "514850861": ["acd-chre_bt_offload_data_rx"],
    "514850860": ["acd-chre_bt_offload_data_tx"],
    "213909586": ["ttySAC18"]
  },
  "camera": {
    "260046860": ["dma_heap/video_system"],
    "260046861": ["dma_heap/video_system-uncached"],
    "510656512": ["gxp"],
    "84934660": ["video10"],
    "84934661": ["video11"],
    "84934662": ["video12"],
    "84934656": ["video6"],
    "84934657": ["video7"],
    "84934658": ["video8"],
    "84934659": ["video9"],
    "509607953": ["lwis-act-cornerfolk"],
    "509607954": ["lwis-act-cornerfolk-dokkaebi"],
    "509607955": ["lwis-act-cornerfolk-taotie-uw"],
    "509607942": ["lwis-be-core"],
    "509607959": ["lwis-dpm"],
    "509607950": ["lwis-eeprom-djinn"],
    "509607952": ["lwis-eeprom-smaug-dokkaebi"],
    "509607951": ["lwis-eeprom-smaug-taotie-uw"],
    "509607957": ["lwis-flash-lm3644"],
    "509607938": ["lwis-gdc0"],
    "509607939": ["lwis-gdc1"],
    "509607941": ["lwis-gse"],
    "509607944": ["lwis-gtnr-align"],
    "509607943": ["lwis-gtnr-merge"],
    "509607937": ["lwis-isp-fe"],
    "509607945": ["lwis-lme"],
    "509607940": ["lwis-mcsc"],
    "509607956": ["lwis-ois-djinn"],
    "509607947": ["lwis-sensor-boitata"],
    "509607948": ["lwis-sensor-dokkaebi"],
    "509607949": ["lwis-sensor-taotie-uw"],
    "509607958": ["lwis-slc"],
    "509607960": ["lwis-test"],
    "509607936": ["lwis-top"],
    "509607946": ["lwis-votf"]
  },
  "gnss": {
    "521142293": ["gnss_boot"],
    "521142294": ["gnss_dump"],
    "10485880": ["gnss_ipc"],
    "523239425": ["sscd_gnss"]
  },
  "audio_in": {
    "121634827": ["pcmC0D10c"],
    "121634828": ["pcmC0D11c"],
    "121634829": ["pcmC0D12c"],
    "121634830": ["pcmC0D13c"],
    "121634832": ["pcmC0D15c"],
    "121634834": ["pcmC0D17c"],
    "121634837": ["pcmC0D20c"],
    "121634838": ["pcmC0D21c"],
    "121634839": ["pcmC0D22c"],
    "121634843": ["pcmC0D26c"],
    "121634844": ["pcmC0D27c"],
    "121634851": ["pcmC0D32c"],
    "121634853": ["pcmC0D54c"],
    "121634825": ["pcmC0D8c"],
    "121634826": ["pcmC0D9c"]
  },
  "nfc": {
    "10485853": ["st21nfc"]
  }
}
```


Ground-Truth Nodes


```
{
  "camera": {
    "508559371": ["lwis-sensor-boitata"],
    "508559373": ["lwis-sensor-taotie-uw"],
    "508559370": ["lwis-votf"],
    "508559366": ["lwis-be-core"],
    "508559361": ["lwis-isp-fe"],
    "508559377": ["lwis-act-cornerfolk"],
    "508559383": ["lwis-dpm"],
    "508559362": ["lwis-gdc0"],
    "508559380": ["lwis-ois-djinn"],
    "508559381": ["lwis-flash-lm3644"],
    "508559364": ["lwis-mcsc"],
    "508559379": ["lwis-act-cornerfolk-taotie-uw"],
    "508559360": ["lwis-top"],
    "508559367": ["lwis-gtnr-merge"],
    "508559368": ["lwis-gtnr-align"],
    "508559382": ["lwis-slc"],
    "508559372": ["lwis-sensor-dokkaebi"],
    "508559378": ["lwis-act-cornerfolk-dokkaebi"],
    "508559365": ["lwis-gse"],
    "508559363": ["lwis-gdc1"],
    "84934656": ["video6"],
    "84934657": ["video7"],
    "84934662": ["video12"]
  },
  "bluetooth": {
    "213909586": ["ttySAC18"]
  },
  "nfc": {
    "10485854": ["st21nfc"]
  },
  "audio_in": {
    "121634825": ["pcmC0D8c"]
  },
  "gnss": {
    "521142294": ["gnss_dump"],
    "10485880": ["gnss_ipc"]
  }
}
```